"NewSQL"

New Architectures

- Classic DBMS architecture has a long history going back to 1970s
  - Though many improvements over the decades
- Last decade: rise of "NoSQL" alternatives
  - Pros: high scalability, flexible data model
  - Cons: no joins, no ACID transactions, eventual consistency

"NewSQL"

- Recent trend: systems that retain some features provided by RDBMSs
  - SQL-like languages
  - Transactions
- But are radically redesigned for today's hardware
  - Lots of main memory
  - Systems are distributed/replicated by design rather than as an afterthought

Customization

- Custom solutions and architectures for particular classes of applications
- OLTP workloads: Online Transaction Processing
- OLAP workloads: Online Analytics Processing

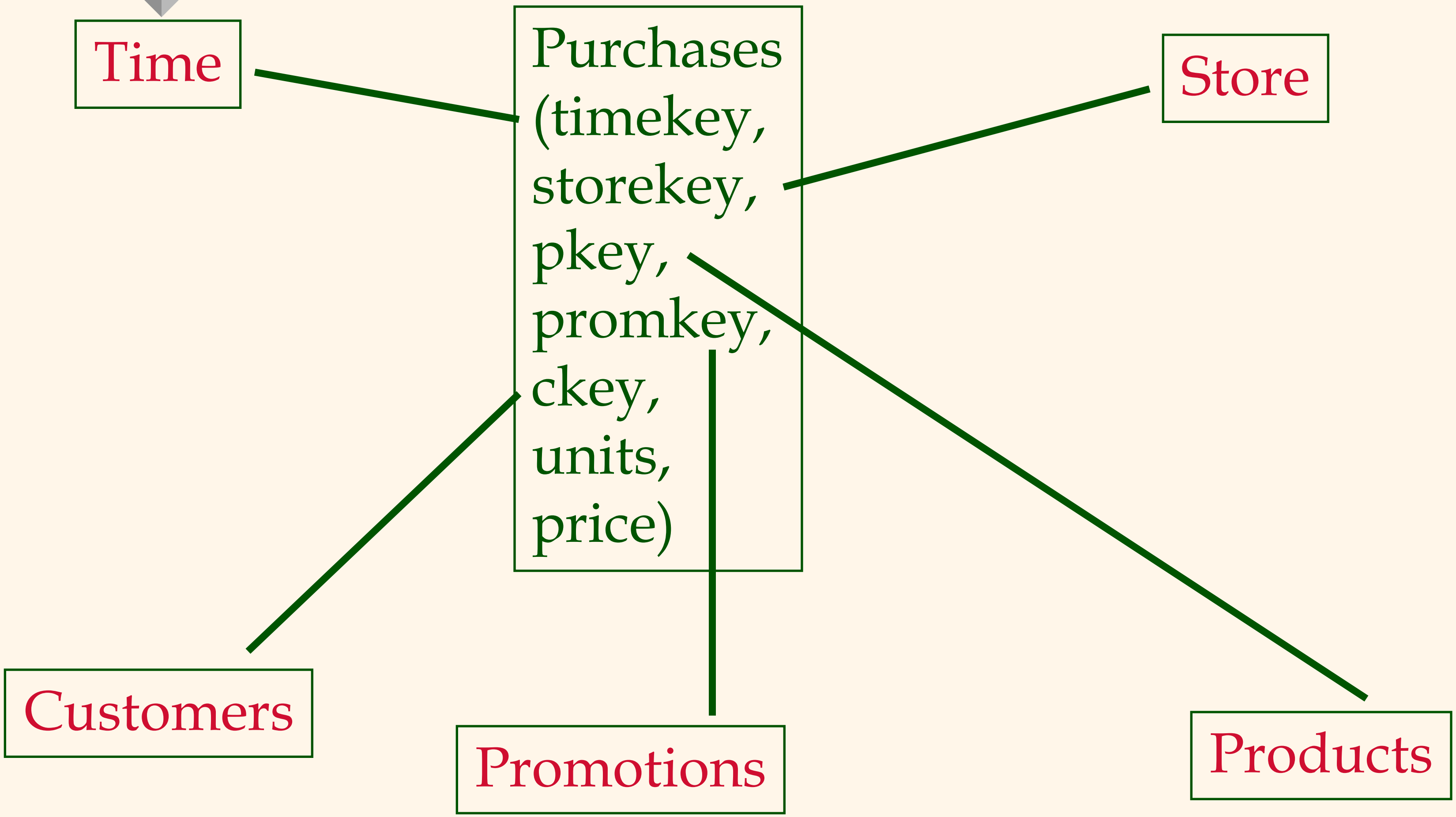
The OLTP database

- An OLTP database stores the current snapshot of the business:
  - Current customers with current addresses
  - Current inventory
  - Current orders
  - Current account balance

The Data Warehouse (OLAP DB)

- Historical collection of all relevant data for analysis purposes
- Examples:
  - Current customers versus all customers
  - Current orders versus history of all orders
  - Current inventory versus history of all shipments
- Stores information that might be useless for the operational part of a business

OLAP Star Schema



OLTP vs. OLAP

	OLTP	OLAP
Typical user	Clerical	Management
System usage	Regular business	Analysis
Workload	Read/Write	Read only
Types of queries	Predefined	Ad-hoc
Unit of interaction	Transaction	Query
Level of isolation required	High	Low
No of records accessed	<100	>1,000,000
No of concurrent users	Thousands	Hundreds
Focus	Data in and out	Information out

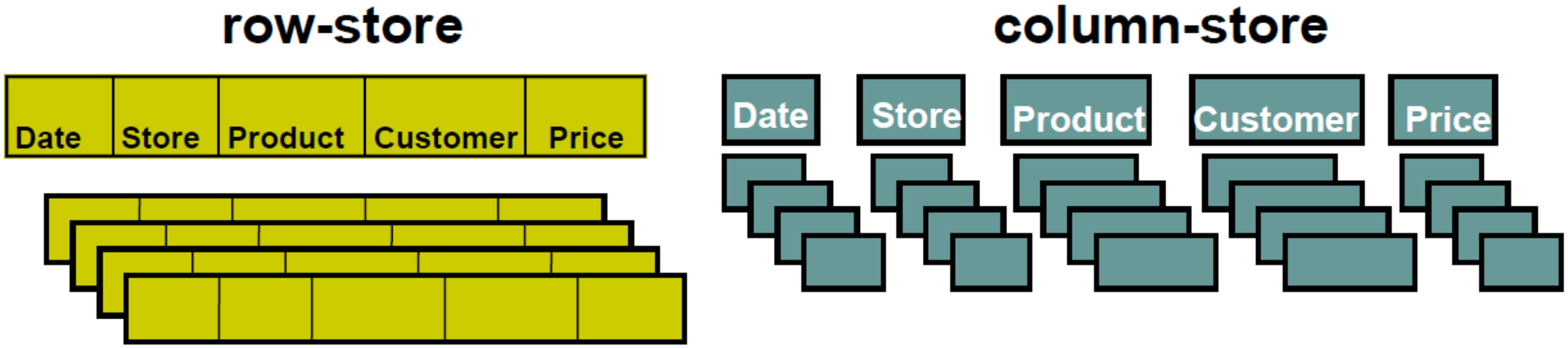
Today's lecture

- 3 interesting recent systems
- [C-Store] (commercial fork: Vertica)
  - Store tables by columns rather than rows to optimize performance for OLAP queries
- [H-Store] (commercial fork: VoltDB)
  - Custom architecture tuned for OLTP workloads
  - Heavy use of main memory
- [Spanner] (an internal system used at Google)
  - SQL-like language over a key-value store
  - Strong consistency (Paxos!!)

Readings

- C-Store (commercial fork: Vertica)
  - http://db.csail.mit.edu/projects/cstore/vldb.pdf
- H-Store (commercial fork: VoltDB)
  - http://hstore.cs.brown.edu/papers/hstore-endofera.pdf
- Spanner (an internal system used at Google)
  - http://research.google.com/archive/spanner.html

Column Stores



+ easy to add/modify a record

- might read in unnecessary data

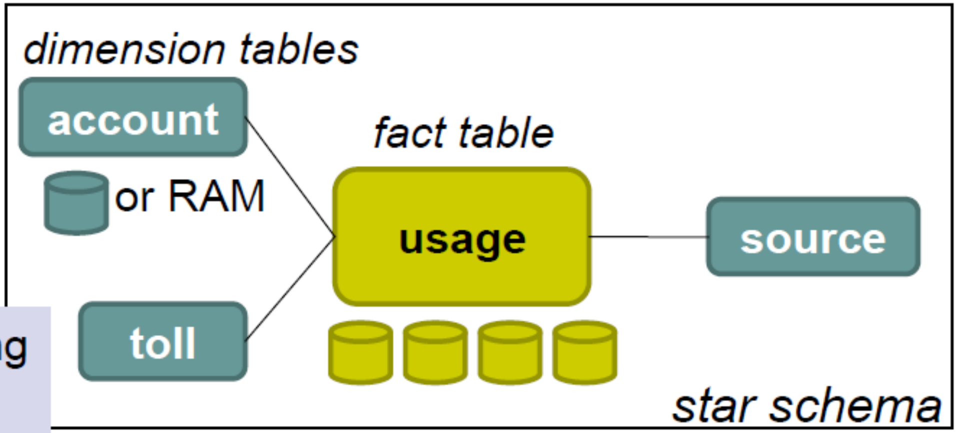
+ only need to read in relevant data

- tuple writes require multiple accesses

=> *suitable for read-mostly, read-intensive, large data repositories*

Data Warehousing example

1 Typical DW installation



1 Real-world example

"One Size Fits All? - Part 2: Benchmarking Results" Stonebraker et al. CIDR 2007

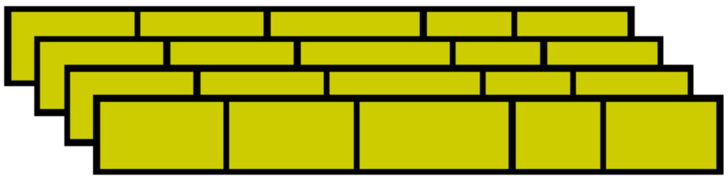
```
QUERY 2
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE'. 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

	Column-store	*Row-store*
Query 1	*2.06*	*300*
Query 2	*2.20*	*300*
Query 3	*0.09*	*300*
Query 4	*5.24*	*300*
Query 5	*2.88*	*300*

Why? Three main factors (next slides)

Example Explained

row store



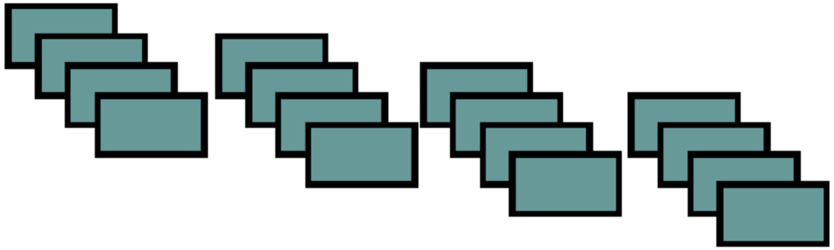
read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

What about vertical partitioning? (it does not work with ad-hoc queries)

column store



read only columns needed

in this example: 7 columns

caveats:

- "select * " not any faster
- clever disk prefetching
- clever tuple reconstruction

Compression helps too

- Columns compress better than rows
- Contain values from a single domain, significantly less entropy:
  - Male, Female, Female, Male, Male, Male,
  - 1998, 1998, 1998, 1999, 1999, 2000, 2001,…

What does C-Store contain?

- Relational model with tables
- But tables not stored directly at all
- Instead system stores projections, i.e. Materialized Views (MVs)
- Some number of columns from a table
  - maybe from more than one table if foreign key-primary key relationship
- Sorted by one of the attributes

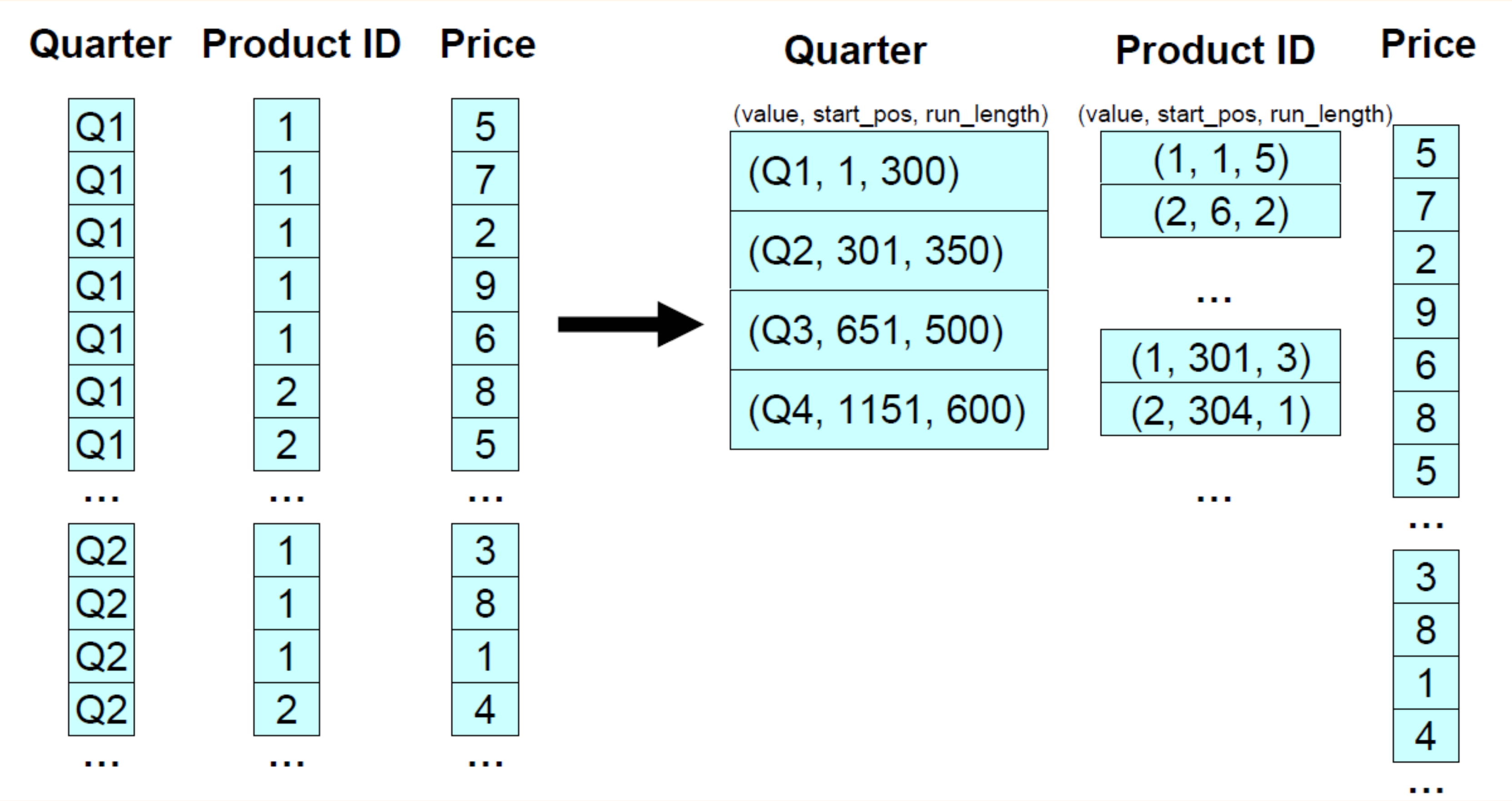
Example

User view:	Possible set of MVs:
EMP (name, age, salary, dept)	MV-1 (name, dept, floor) in floor order
Dept (dname, floor)	MV-2 (salary, age) in age order
	MV-3 (dname, salary, name) in salary order

Join Indexes

- Note that we must be careful because decomposing into columns is not lossless join in general
- So need to maintain join indexes to allow reconstituting a tuple if needed
- Note that we do NOT store artificial keys/tuple identifiers, to save space
  - The storage key of an entry in a projection is just its position/offset

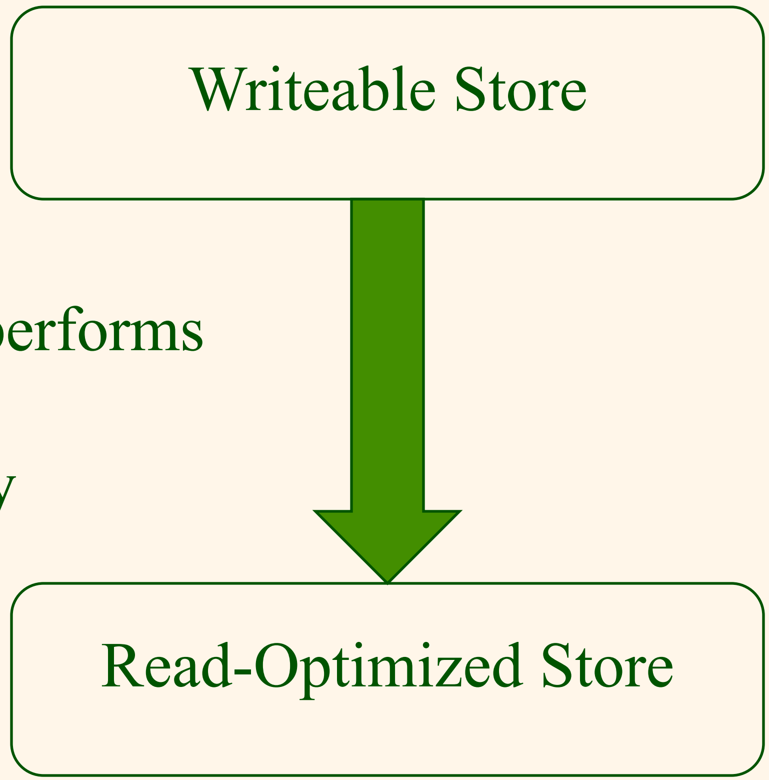
Compressing data



Running queries

- New algorithms and evaluation plans needed
- Operate directly on compressed data vs decompress
- If need attributes that are available in multiple projections, which ones should be chosen?

Handling writes is trickier



Tuple mover performs batch inserts asynchronously

Column Store systems "for real"

- C-store was commercialized as Vertica
- Another system: MonetDB
- Many column store systems by various vendors for fast analytics

H-Store/VoltDB

- What about OLTP workloads?
- We can improve performance on these too!
- Exploit modern hardware
- Exploit unique features of OLTP workloads
- Example: H-Store (comercialized as VoltDB)

Today's OLTP workloads

- Data is relatively small, < 100 GB typically
- Transactions are short-running, no "user stalls"
  - This is no longer the 1970s where you input SQL at a terminal
  - When you buy something on Amazon, typically split into several underlying transactions

Today's computers

- Memory is no longer tiny
  - 100GB of RAM? Sure, you can outfit a machine with that
- Your database no longer sits on a single box
  - Have available infrastructure that is distributed and replicated for fault-tolerance

H-Store design decisions

- Run everything in memory
  - Could rely on replication and failover for durability
    - So, only need to log for undo purposes (not for redo)
  - Though VoltDB does use periodic disk snapshots

H-Store design decisions

- Run all transactions serially
  - Hey, they're short anyway
- Result: saves on some major overhead
  - Disk accesses
  - Synchronization/concurrency

OLTP Workloads

- Make use of the fact that OLTP workloads are not ad-hoc
- Require all possible transaction classes to be predefined and registered with the system
  - Can be pre-optimized
  - For distributed transactions, can identify which of them really require inter-site communication/2PC
- Allows a better DB design as we know the entire workload up front (data partitioning etc.)

Summary so far

- Custom solutions and architectures for particular classes of applications
- Column stores for read-mostly, OLAP style workloads
- H-Store and similar systems for OLTP workloads
  - In-memory
  - Transactions run serially
  - Optimized for a fully pre-specified workload

Google Spanner

- Distributed multiversion database
  - General-purpose transactions (ACID)
  - SQL query language
  - Schematized tables
  - Semi-relational data model
- Running in production
  - Storage for Google's ad data
- Presented at OSDI (major systems conference) in 2012

Overview

- Supports lock-free distributed read transactions
- Guarantees external consistency (linearizability) of distributed transactions
  - A combination of serializability and linearizability
  - If T1 commits before T2 starts, then T1 is serialized before T2
  - First system at global scale to enforce this

Read Transactions

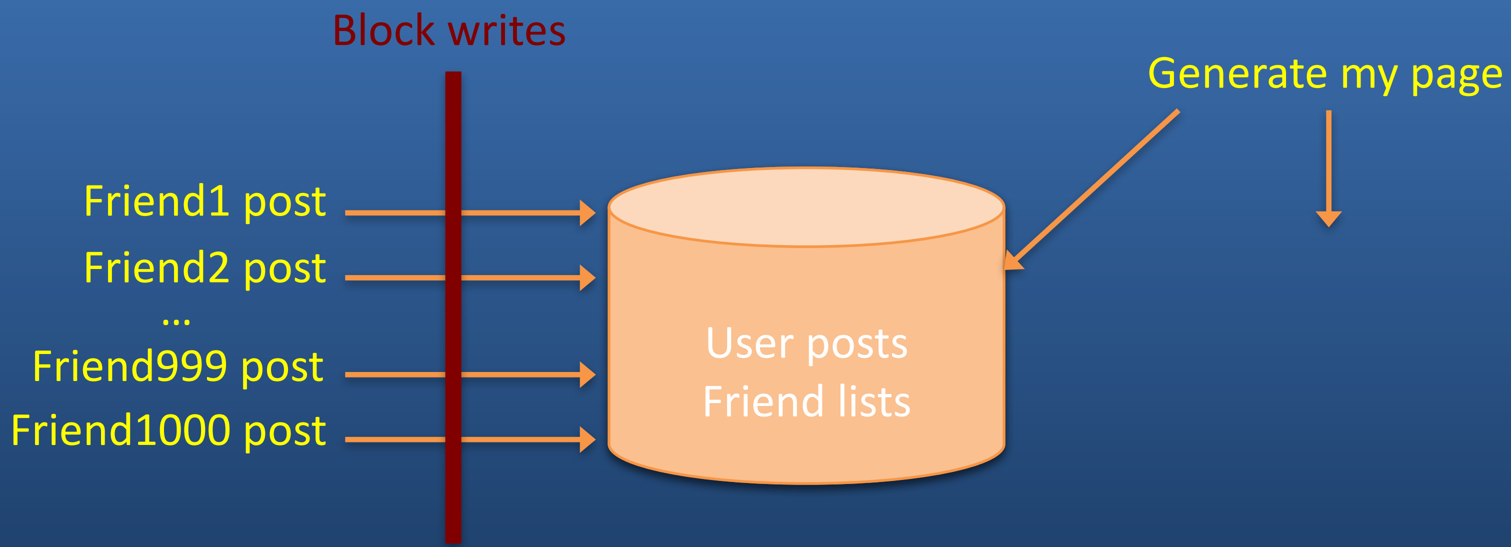
- In a social network, generate a page of friends’ recent posts
  - Consistent view of friend list and their posts

Why consistency matters

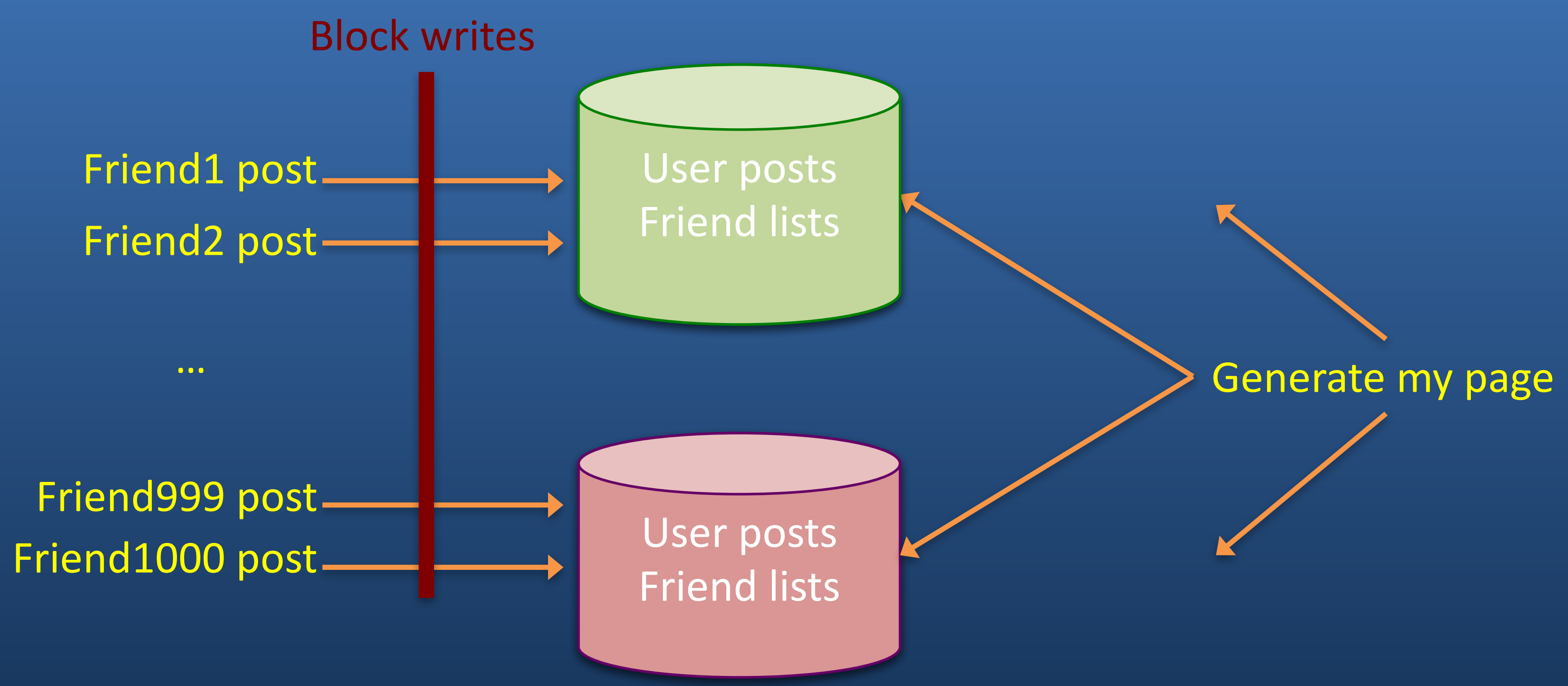
1. Remove untrustworthy person X as friend
2. Post P: “My government is repressive…”

Google

Single Machine



Multiple Machines



Version Management

- Each writer transaction *T* assigned a timestamp *s*
- Data written by *T* is timestamped with *s*

Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Synchronizing snapshots

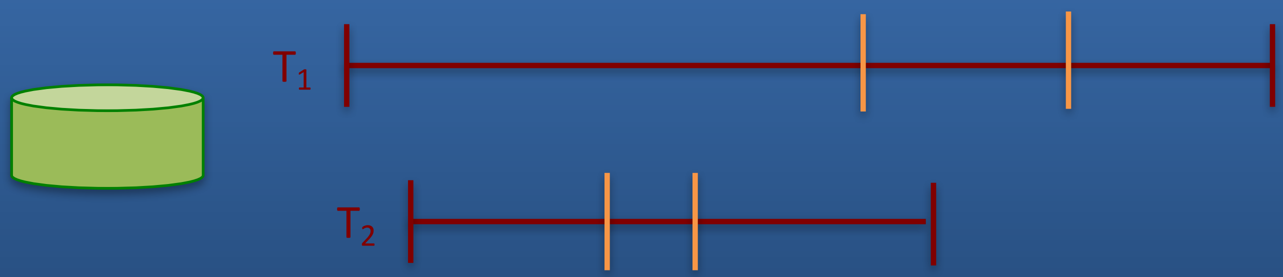
- Implementation relies on appropriate use of transaction timestamps

Assigning Timestamps

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held

Some Timestamp Guarantees

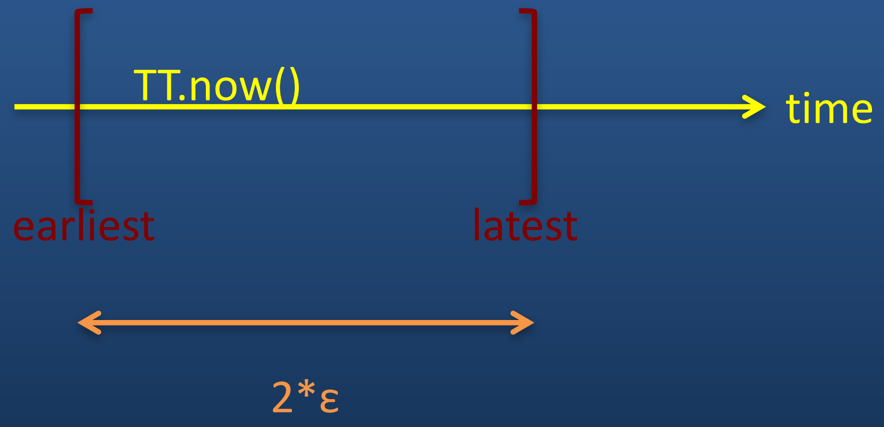
- For conflicting transactions, timestamp order == serialization order



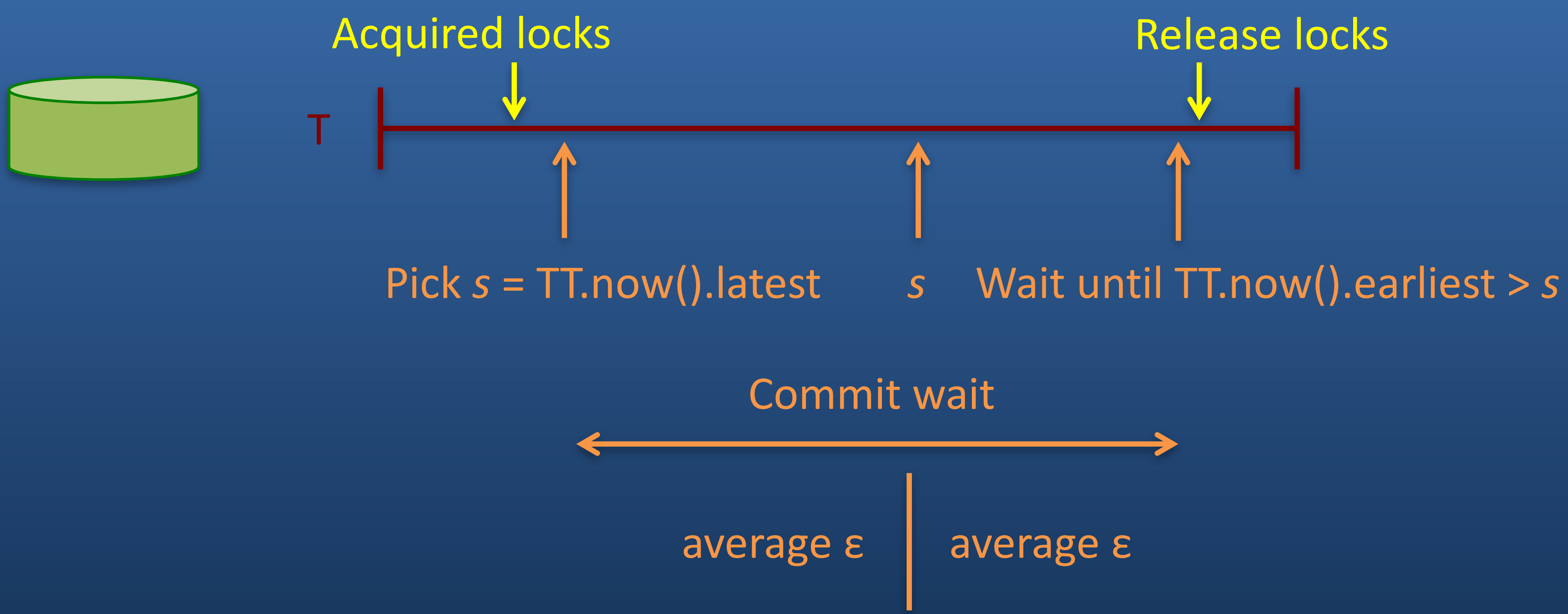
- T4 starts after T3 ends => T4 has smaller timestamp

TrueTime API

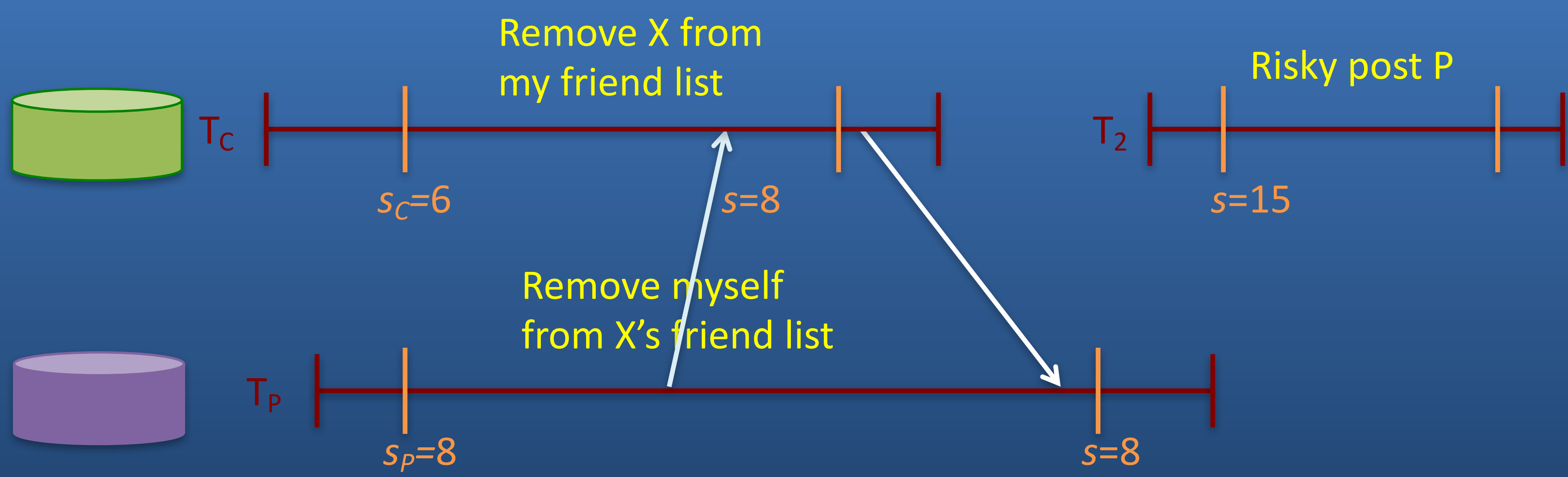
- "Global wall-clock time" with bounded uncertainty



Timestamps and TrueTime



Timestamps for Distributed Xacts



Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Summary

- Lock-free read transactions across datacenters
- External consistency
  - A very strong formal guarantee
- TrueTime
  - Uncertainty in time can be waited out
- More details (e.g. how to actually implement consistent reads at a time/version) in paper

Slide credits

- VLDB 2009 tutorial on Column stores
  - http://www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf
- H-Store slides
  - http://hstore.cs.brown.edu/slides/hstore-vldb2007.pdf
- Google Spanner Slides
  - http://research.google.com/archive/spanner-osdi2012.pptx